VIRTUAL MEMORY

GU/Jianhua
谷建华
School of Computer Science
Northwestern Polytechnical University

Virtual Memory

- Background
- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Page-Replacement Algorithms
- Allocation of Frames
- Thrashing
- Other Considerations
- Demand Segmentation

Background

- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.
- Benefits:
  - Program size not constrained by amount of physical memory available.
  - Programmer do not consider the amount of physical memory.
  - More programs can be run simultaneously
  - Less need for swapping
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Issues need resolved

- Can a program execute correctly if part of program are loaded into memory?
- How can OS do if process access an data or instruction that is not in memory?
- How can the OS do if there is not enough free memory in main memory?

OS Policies for Virtual Memory(1)

- Fetch Policy
  - How/when to get pages into physical memory.
  - demand paging *vs*. pre-paging.
- Placement Policy
  - Where in physical memory to put pages.
- Replacement Policy
  - Physical memory is full. Which frame to page out?

OS Policies for Virtual Memory(2)

- Resident Set Management Policy
  - How many frames to allocate to process?
- Cleaning Policy
  - When to write a modified page to disk.
- Load Control
  - How many processes will be resident in main memory, which is referred to as the multiprogramming level?

Demand Paging

- The principle of locality of reference.
- Page Fault Processing
- Replacement

The Principle of Locality of Reference

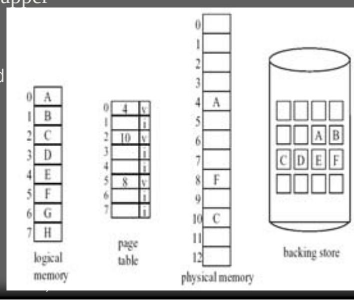
- A program that references a location *n* at some point in time is likely to reference the same location *n* and locations in the immediate vicinity of *n* in the near future.
- As a process executes, it move from locality to locality.

程序局部性原理

- 程序局部性原理是指程序在执行时呈现出局部性规律，即在一较短时间内，程序的执行仅限于某个部分，相应地，它所访问的存储空间也局限于某个区域。
- 局部性又表现为*时间局部性*和*空间局部性*。
- 时间局部性是指如果程序中的某条指令被执行，则不久以后该指令可能再次执行。如果某数据结构被访问，则不久以后该数据结构可能再次被访问。
- 空间局部性是指一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问。

Demand Paging

- Bring a page into memory only when it is needed (not in memory).
- Also called Lazy Swapper
- Benefits:
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users



Page Table Entry

- Page table entry:

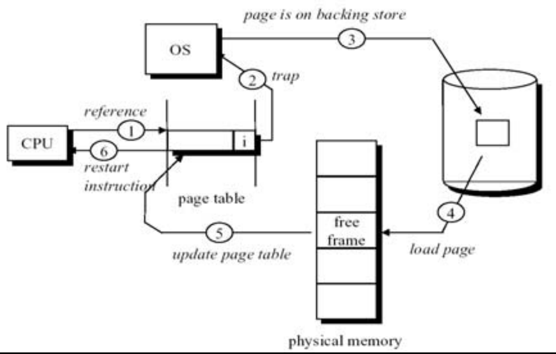
V	M	C	Swap Address	Frame Number

- V: Valid-Invalid Bit
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
  Initially valid–invalid bit is set to 0 on all entries.
- M: Modified Bit
- C: Control Bit

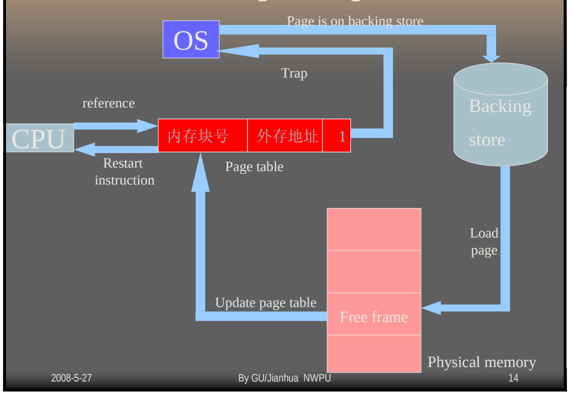
Page Fault

- *Page fault* : During address translation, if valid–invalid bit in page table entry is 0（Page NOT in memory）.
- *Page fault trap*: If there is ever a reference to a page, the reference will trap to OS when page fault
- OS looks at another table to decide:
  - Invalid reference(out of process space) ⇒ abort.
  - Just not in memory.
- Find a free frame.
- Swap page into frame.(read page from swap)
- Reset tables: set validation bit = 1.
- Restart instruction

Processing of Page Fault



Processing Of Page Fault



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.
- Use *modify (dirty) bit* to reduce overhead of page transfers – only modified pages are written to disk.

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – p) x memory access + p x page_fault time

page_fault time = (service page fault interrupt
+ [swap page out]
+ swap page in
+ restart overhead)

Demand Paging Example

- Memory access time = 100 nanoseconds
- Average page-fault time = 25 milliseconds
- EAT = (1 – p) x 100 + p x (25 000 000)
  = 100+24 999 900 x p
  - When p=0.1%, EAT=25 microseconds
  - If we want less than 10% degradation,
    100+100 x 10% >100 + 25 000 000 x p
    p < 0.000 000 4

Page-Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- In all our examples, the reference string is
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- Algorithms
  - Optimal Algorithm
  - FIFO Algorithm
  - Least Recently Used (LRU) Algorithm
  - Random Algorithm

Optimal Algorithm

- Replace page that will not be used for longest period of time in the future.
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

- How do you know this? (Prediction)
- Used for measuring how well your algorithm performs.

First-In-First-Out (FIFO) Algorithm

- Replace page that has been in memory the longest(oldest)
- 3 frames (3 pages can be in memory at a time per process)

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	4	5
2	2	1	3
3	3	2	4

9 page faults

- 4 frames

1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

10 page faults

- FIFO Replacement – Belady's Anomaly
  - more frames ⇏ less page faults

Least Recently Used (LRU) Algorithm

- Replace page in memory that has not been referenced for longest time.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	5	
2		
3	5	4
4	3	

- Basic Algorithm:
  - Counter implementation
  - Stack implementation

Replacement Example

- Demo

LRU: Counter implementation

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the (logical) clock into the counter.
  - When a page needs to be changed, look at the counters to determine which page are to change.
  - the LRU page with the smallest time values
- Features
  - Request a search of page table to find the LRU page
  - Write the page table whenever each memory access
  - Overflow of clock must be considered

LRU: Stack implementation

- Stack implementation – keep a stack of page numbers in a double link list:
  - Page referenced:
    - Remove it from stack and put it on the top
    - Top page is always the most used page and bottom is *LRU* page
- Features
  - requires 6 pointers to be changed at worst when moving page
  - No search for replacement
- Demo

LRU Approximation Algorithms

- Basic Idea:
  - Reference bit
    - With each page associate a bit, initially = 0
    - When page is referenced, the bit is set to 1.
    - Replace the one whose bit is 0 (if one exists). We do not know the order, however.
- Algorithms:
  - Additional-Reference-Bits Algorithm
  - Second-Chance Algorithm
  - Enhanced Second-Chance Algorithm

LRU: Additional-Reference-Bits Algorithm

- Keep an 8-bit *shift register* for each page
- At regular intervals(say, every 100m), OS shifts the reference bit for each page into high-order bit of shift register, shifting the other bit right 1 bit. RB → Shift Register
- If we interpret shift register as unsigned integers, the page with the lowest number is the LRU page, and it can be replace.
- *Note: The number are not guaranteed to be unique. We can either replace all page with smallest value, or user FIFO selection.*
- Example: page with 11000100 has been used more recently than one with 01110111

LRU: Second-Chance Algorithm

- Need one reference bit for each page
- Algorithm:

repeat to inspect reference bit for each page:

If (reference bit is 0) replace that page;

else { set reference bit 0;
leave page in memory;
move on next page;
loop; }

LRU: Enhanced Second-Chance Algorithm

- Both reference bit and modify bit are needed, which is an ordered pair <R,M>
- Algorithm:
  - If(<0,0>) page is selected;
  - If(<0,1>) set the *M bit* to 0; and move on(next);
  - If(<1,0>) set the *R bit* to 0; and move on;
  - If(<1,1>) set the *M bit* to 0; and move on;

Page-Buffering (1)

- Issue: cost of replacing a page that has been modified is greater than for one that has not.
- The replaced page is not swapped out, but rather is held in memory.
- System maintain two list: free page list and modified page list in memory.
  - If(no modified) put it into free list;
  - If(modified) put it into modified list;

Page-Buffering (2)

- When a page fault occurs, OS first check whether the desired page is in the free or modified list.
- These replaced pages are reused as soon as possible.
- Modified pages are written out in cluster rather than one at a time.
- Size of the two lists is fixed.
- FIFO algorithm is used to manage the list.

Allocation of Frames

- Each process needs minimum number of pages. As the number of frames allocated to each process decreases, the page fault-rate increases, slowing process execution.
- How many frames does each process get from the fixed amount of free memory ?
- The minimum number of frames that must be allocated to a process is defined by instruction-set architecture.
- Allocation schemes.
  - Fixed allocation
  - Proportional allocation
  - Priority allocation

Fixed Allocation

- Equal allocation – To split *m* frames among *n* processes and to give each process equal share, *m*/*n*
- Example: if 100 frames and 5 processes, give each 20 pages.

Proportional allocation

- Proportional allocation – Allocate according to the size of process.

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.
- If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

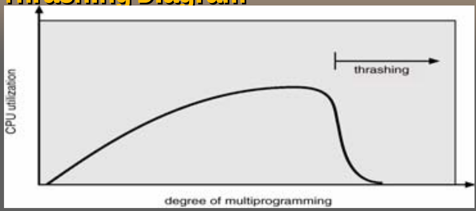
Global vs. Local Replacement

- *Global replacement* – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- *Local replacement* – each process selects from only its own set of allocated frames.

Thrashing

- If a process does NOT have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- Thrashing ≡ a process is busy swapping pages in and out.
- A process is spending more time paging than executing.

Thrashing Diagram



- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

Working-Set Model (1)

- Working Set: *WS(t, Δ)* : set of pages referenced by process during time interval *(t- Δ, t)*
- Δ ≡ working-set window ≡ a fixed number of page references
  Example: 10,000 instruction
- Page reference table

… 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4.

Δ t1 WS(t1)={1,2,5,6,7}

Δ t2 WS(t2)={3,4}

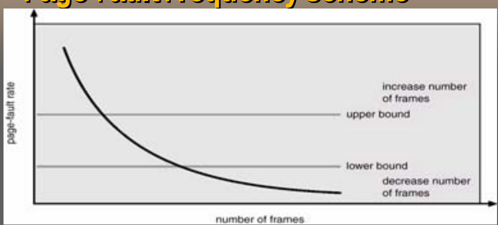
Working-Set Model (2)

- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if Δ = ∞ ⇒ will encompass entire program.
- $D = \Sigma WSS_i \equiv$ total demand for frames
- if $D > m$ ⇒ Thrashing (*m is total of physical frame)*
- Policy if *D* > m, then suspend one of the processes

Keeping Track of the Working Set

- Approximate with *interval timer* + *a reference bit*
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 ⇒ page in working set.
- This is not completely accurate.
- Improvement = 10 bits and interrupt every 1000 time units.

Page-Fault Frequency Scheme



- Establish "acceptable" page-fault rate.
  - If actual rate too low, process may have too many frame.
  - If actual rate too high, process needs more frame.
  - If(rate > upper limit) allocate process another frame
  - If(rate < lower limit) remove frame from process

Demand Segmentation

- Used when insufficient hardware to implement demand paging.
- OS allocates memory in segments, which it keeps track of through segment descriptors
- Segment descriptor contains a valid bit to indicate whether the segment is currently in memory.
  - If segment is in main memory, access continues,
  - If not in memory, segment fault.

请求分段的段表机制

- 段表项
  - 段名，段长，基址，访问方式，访问位，修改位，内存标志，外存地址
- 地址变换/缺段终端
- 以段为单位换入/换出，开销比较大。

Other Consideration

- Self-Study